Examen de Programación II
(Ingeniería Informática)

Junio 2009

1) Lenguaje C (2 puntos)

Escribir el módulo "Arrays" (ficheros `arrays.h` y `arrays.c`) que permita realizar las siguientes operaciones con arrays de números enteros:

- `cuenta_ocurrencias`: cuenta las veces que aparece un determinado número en un array. Recibe como parámetros de entrada el array y el número. Devuelve en un parámetro de salida el número de ocurrencias del número en el array. Retorna 0 si el número no está ninguna vez en el array y 1 si el número está una o más veces en el array.
- `copia`: realiza la copia de un array. Recibe como parámetro de entrada el array a copiar y retorna un nuevo array que es una copia del anterior (tiene su misma longitud y sus mismos elementos).

(Ambas funciones (`cuenta_ocurrencias` y `copia`) podrían requerir un parámetro de entrada adicional no especificado anteriormente).

Escribir un programa principal muy sencillo (que no pida datos al usuario) que utilice las dos funciones del módulo "Arrays".

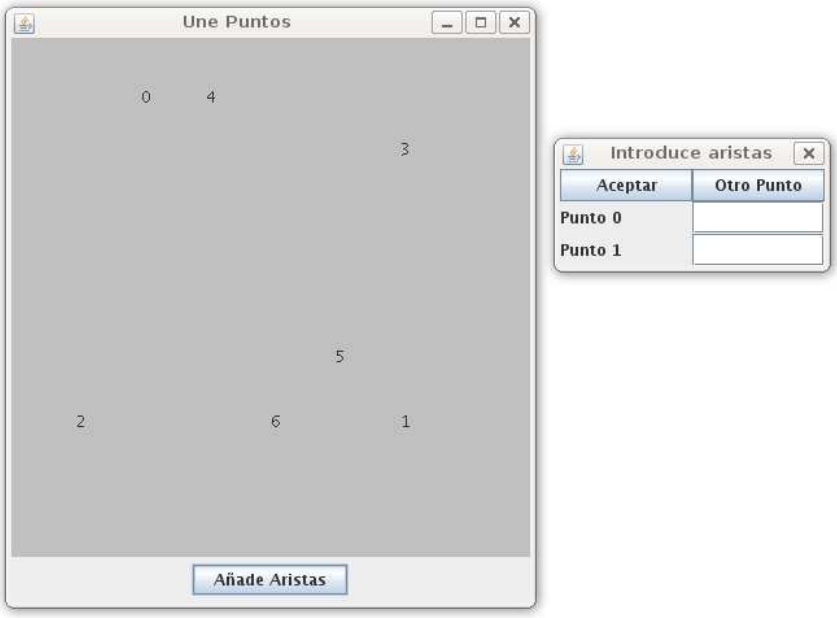
Solución:

```
/*****************************************************************
 * arrays.h
 */

/*
 * Cuenta las veces que aparece un determinado número en un array
 * Parámetros:
 *   a - array en el que contar las repeticiones
 *   largo - número de elementos del array a
 *   n - número del que contar las repeticiones
 *   num_rep - repeticiones del número 'n' en el array 'a'
 * Retorna:
 *   1 si el número 'n' aparece al menos una vez en el array 'a'
 *   0 si el número 'n' no está en el array 'a'
 */
int cuenta_repeticiones(const int a[], int largo, int n,
                        int *num_rep);

/*
 * Realiza la copia de un array
 * Parámetros:
 *   original - array a copiar
 *   largo - número de elementos del array 'original'
 * Retorna:
 *    Nuevo array que es una copia del original (tiene su misma
 *   longitud y sus mismos elementos).
 */
int * copia(const int original[], int largo);
```

```c
/*************************************************************
 * arrays.c
 */
#include <stdlib.h>
#include "arrays.h"

/*
 * Cuenta las veces que aparece un determinado número en un array
 */
int cuenta_repeticiones(const int a[], int largo, int n,
                        int *num_rep) {
  *num_rep=0;
  for(int i=0; i<largo; i++)
    if (a[i]==n)
      (*num_rep)++;

  if (*num_rep==0)
    return 0;
  else
    return 1;
}

/*
 * Realiza la copia de un array
 */
int * copia(const int original[], int largo) {
  // reserva el espacio de memoria correspondiente a un array de
  // igual tamaño que el original
  int *nuevo = (int *)calloc(largo, sizeof(int));

  // copia cada elemento del original en el nuevo
  for(int i=0; i<largo; i++)
    nuevo[i]=original[i];

  return nuevo;
}



/*************************************************************
 * Programa de prueba del módulo "Arrays"
 */
#include <stdio.h>
#include <stdlib.h>
#include "arrays.h"

/*
 * Muestra un array por consola (no pedido en el examen)
 */
void muestra_array(const int a[], int largo) {
  printf("[");
  for(int i=0; i<largo; i++) {
    printf("%d", a[i]);
    if (i < largo-1)
```

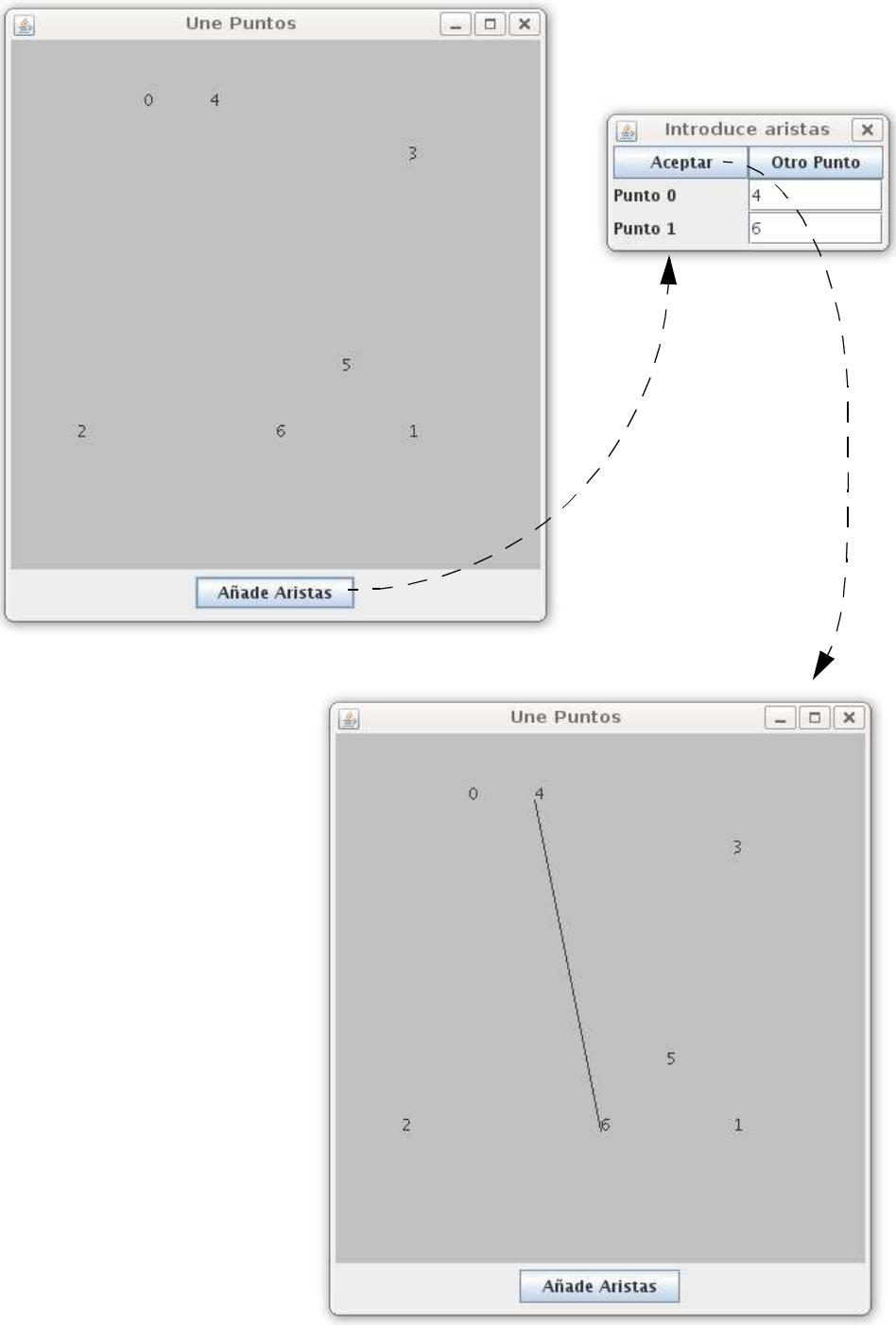
```
        printf(", ");
    }
    printf("]");
}

/*
 * Programa sencillo de prueba del módulo "Arrays"
 */
int main(void) {
    int a1[] = {1,2,3,1,4,2,1};
    int buscado = 1;
    int num_repeticiones;
    int *a2;

    // cuenta el número de repeticiones
    int encontrado = cuenta_repeticiones(a1, 7, buscado,
                                         &num_repeticiones);

    printf("Ocurrencias de %d en a1:%d (encontrado:%d)\n", buscado,
           num_repeticiones, encontrado);

    // copia a1
    a2 = copia(a1, 7);
    printf("Copia de a1:\n");
    muestra_array(a2,7);

    return 0;
}
```

2) (3.5 puntos) Se pretende realizar una aplicación gráfica formada por la ventana `VentanaUnePuntos` y el diálogo `DialogAñadeArista` que tienen la apariencia inicial mostrada a continuación:

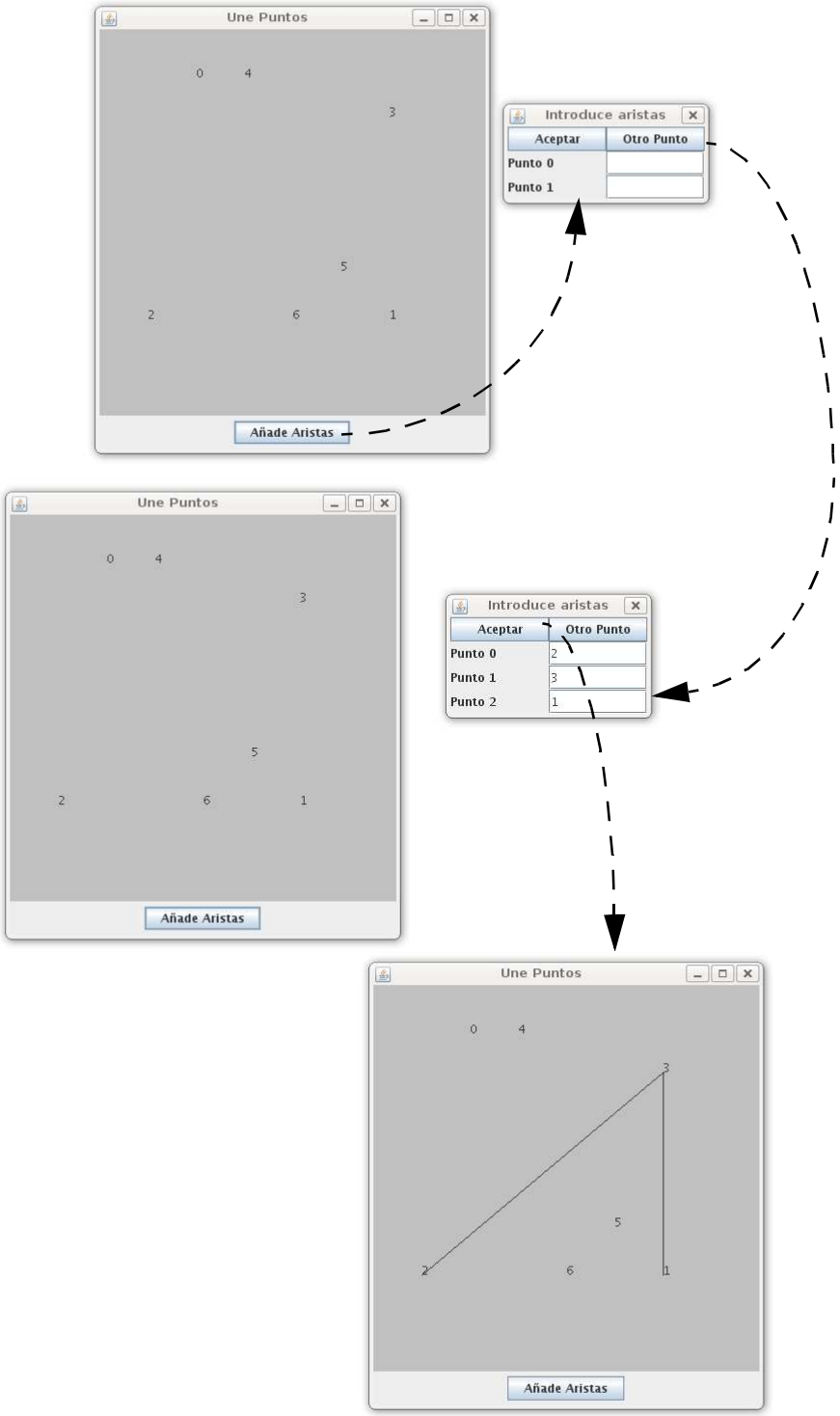


La ventana `VentanaUnePuntos` debe mostrar un conjunto de puntos (en el ejemplo etiquetados del 0 al 6) inicialmente sin unir. Los puntos existentes son fijos y se obtienen con la clase `Puntos` (mostrada más adelante).

La pulsación del botón "Añade Aristas" provoca la aparición del diálogo `DialogAñadeArista` que permite introducir los puntos inicial y final de una arista. La pulsación del botón "Aceptar" después de haber introducido los puntos en el diálogo provocará la aparición de la arista (una línea recta que une los dos puntos). La figura siguiente muestra el proceso de creación de una arista:



El botón "Otro punto" del diálogo `DialogAñadeArista` permite introducir un nuevo punto en el diálogo, lo que permitirá crear otra arista más. El proceso se muestra en la figura siguiente:



La implementación de la funcionalidad descrita para el botón "Otro Punto" se valorará con 1.0 punto de los 3.5 puntos correspondientes a esta pregunta.

Escribir todo el código de la aplicación (excepto la distribución de componentes en la ventana `VentanaUnePuntos`).

Seguir el patrón "Modelo-Vista-Controlador".

Se dispone de las siguientes clases ya realizadas:

```
/**
 * Coordenadas de un punto en pixels
 */
public class Punto {
  public int x;
  public int y;
}
```

```
/**
 * Lista con todos los puntos existentes
 */
public class Puntos {

  private ArrayList<Punto> puntos = new ArrayList<Punto>();

  /**
   * Crea una lista de puntos
   */
  public Puntos() {
    Lee los puntos existentes y les almacena en la lista "puntos"
  }

  /**
   * Retorna el punto que ocupa la posición i-ésima en la lista
   * @param i posición del punto en la lista
   * @return el i-ésimo punto o null en caso de que
   * el valor de i no corresponda a ningún punto
   */
  public Punto punto(int i) {
    if (i<0 || i>=puntos.size())
      return null;
    return puntos.get(i);
  }
}
```

```
/**
 * Arista que une dos puntos
 */
public class Arista {
  public Punto ptoIni; // punto inicial de la arista
  public Punto ptoFin; // punto final de la arista

  public Arista(Punto ptoIni, Punto ptoFin) {
    this.ptoIni = ptoIni;
    this.ptoFin = ptoFin;
  }
}
```

Solución:

```
import java.util.ArrayList;

/**
 * Lista con las aristas
 */
public class Aristas {

  private ArrayList<Arista> aristas = new ArrayList<Arista>();

  /**
   * Añade una arista
   * @param a arista a añadir
   */
  public void añade(Arista a) {
    aristas.add(a);
  }

  /**
   * Retorna la arista i-ésima
   * @param i posición de la arista en la lista
   * @return arista i-ésima o null si la posición no es válida
   */
  public Arista arista(int i) {
    if (i<0 || i>=aristas.size())
      return null;

    return aristas.get(i);
  }
}


import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Ventana que contiene el panel que permite dibujar los puntos
 */
public class VentanaPuntos extends JFrame {

  // modelo
  private Puntos puntos = new Puntos();
  private Aristas aristas = new Aristas();

  // componentes
  private JButton bAñadeAristas = new JButton("Añade Aristas");
  private PanelPuntos panelPuntos = new PanelPuntos(puntos,
                                                    aristas);

  // diálogo para añadir aristas
  private DialogAñadeAristas diálogoAñadeArista =
            new DialogAñadeAristas(this);
```

```
  /**
   * Constructor de la ventana
   */
  public VentanaPuntos() {
    super("Une Puntos");

    // distribuye componentes (no se pedía en el examen)
    // Panel en el centro
    add(panelPuntos, BorderLayout.CENTER);
    // Botón en el sur dentro de un flow
    JPanel pB = new JPanel(new FlowLayout());
    pB.add(bAñadeAristas);
    add(pB, BorderLayout.SOUTH);

    // añade manejadores
    bAñadeAristas.addActionListener(new manejadorBotónAñade());

    // finaliza configuración de la ventana
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pack();
    setVisible(true);
  }

  /**
   * Manejador del botón que permite añadir aristas
   */
  private class manejadorBotónAñade implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
      // muestra el diálogo
      int[] ptos =  diálogoAñadeArista.muestra();

      // si retorna null es porqué el usuario ha cancelado el
      // diálogo o ha ocurrido algún otro problema
      if (ptos != null) {
        for(int i=0; i<ptos.length-1; i++)
          // añade cada arista
          aristas.añade(new Arista(puntos.punto(ptos[i]),
                                   puntos.punto(ptos[i+1])));

        // repinta el panel
        panelPuntos.repaint();
      }
    }
  }

  /**
   * Método main. Crea la ventana principal de la aplicación
   */
  public static void main(String[] args) {
    new VentanaPuntos();
  }
}
```

```
import java.awt.GridLayout;
import java.awt.event.*;
import java.util.LinkedList;
import javax.swing.*;
/**
 * Diálogo que permite introducir las aristas a añadir
 */
public class DialogAñadeAristas extends JDialog {

  // Botones
  private JButton bAceptar = new JButton("Aceptar");
  private JButton bOtroPto = new JButton("Otro Punto");

  // campos de texto y etiquetas: su número no es fijo, ya que se
  // pueden crear con el botón bOtroPto
  private LinkedList<JTextField> camposTexto =
      new LinkedList<JTextField>();
  private LinkedList<JLabel> etiquetas = new LinkedList<JLabel>();

  // almacena los puntos introducidos en el diálogo y retornados
  // a la ventana principal
  private int[] puntos = null;

  /**
   * Constructor
   */
  public DialogAñadeAristas(JFrame owner) {
    super(owner, "Introduce aristas", true);

    setLayout(new GridLayout(0,2));

    // crea los campos de texto y las etiquetas de los 2 primeros
    // puntos
    camposTexto.add(new JTextField());
    etiquetas.add(new JLabel("Punto 0"));
    camposTexto.add(new JTextField());
    etiquetas.add(new JLabel("Punto 1"));

    // distribuye componentes
    add(bAceptar);
    add(bOtroPto);
    add(etiquetas.get(0));
    add(camposTexto.get(0));
    add(etiquetas.get(1));
    add(camposTexto.get(1));

    // añade manejadores
    bAceptar.addActionListener(new ManejadorBotónAceptar());
    bOtroPto.addActionListener(new ManejadorBotónOtroPunto());

    // fin de la configuración del diálogo
    pack();
    setResizable(false);
  }
```

```
/**
 * muestra el diálogo
 * @return puntos introducidos o null si el usuario cancela el
 * diálogo
 */
public int[] muestra() {
  // pone los puntos a null. Se cambia su valor en el manejador
  // del botón aceptar.
  puntos=null;

  // hace visible el diálogo
  setVisible(true);

  // elimina posibles campos de texto creados por el usuario para
  // que la siguiente vez que aparezca el diálogo sólo tenga dos
  while (camposTexto.size() > 2) {
    remove(camposTexto.removeLast());
    remove(etiquetas.removeLast());
  }
  pack(); // ajusta el diálogo a los nuevos componentes

  // cuando se cierra los puntos introducidos están en "puntos"
  return puntos;
}

/**
 * Manejador del botón aceptar
 */
private class ManejadorBotónAceptar implements ActionListener {

  @Override
  public void actionPerformed(ActionEvent e) {
    puntos = new int[camposTexto.size()];
    int i=0;
    try {
      for(i=0; i<camposTexto.size(); i++)
        puntos[i] =
            Integer.parseInt(camposTexto.get(i).getText());
    } catch (NumberFormatException except) {
      // tratamiento de errores (no exigido en el examen)
      puntos = null;
      JOptionPane.showMessageDialog(null, "Error en punto "+i,
          "Error en los datos", JOptionPane.ERROR_MESSAGE);
      return; // NO hace invisible el diálogo
    }

    // hace invisible el diálogo
    setVisible(false);
  }
}
```

```
  /**
   * Manejador del botón otro punto
   */
  private class ManejadorBotónOtroPunto implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
      // añade otra etiqueta y otro campo de texto
      JLabel l = new JLabel("Punto "+camposTexto.size());
      etiquetas.add(l);
      add(l);
      JTextField tf = new JTextField();
      camposTexto.add(tf);
      add(tf);

      // ajusta el tamaño del diálogo para que se vean los
      // nuevos componentes
      pack();
    }
  }
}
```

```
import java.awt.*;
import javax.swing.JPanel;

/**
 * Panel que muestra los puntos y las aristas que los unen
 */
public class PanelPuntos extends JPanel {

  // referencias a las listas de puntos y aristas
  private Puntos puntos;
  private Aristas aristas;

  /**
   * Crea el panel
   * @param puntos lista de puntos a dibujar
   * @param aristas lista de aristas a dibujar
   */
  public PanelPuntos(Puntos puntos, Aristas aristas) {
    this.puntos = puntos;
    this.aristas = aristas;

    setPreferredSize(new Dimension(400, 400));
    setBackground(Color.LIGHT_GRAY);
  }
```

```
    /**
     * sobreescribe el método paintComponent para dibujar los puntos
     * y las aristas
     */
    @Override
    public void paintComponent(Graphics g) {
      super.paintComponent(g);

      // dibuja los puntos
      int i=0;
      Punto p;
      while ((p = puntos.punto(i)) != null) {
        g.drawString(String.valueOf(i), p.x, p.y);
        i++;
      }

      // dibuja las aristas
      i=0;
      Arista a;
      while ((a = aristas.arista(i)) != null) {
        g.drawLine(a.ptoIni.x, a.ptoIni.y, a.ptoFin.x, a.ptoFin.y);
        i++;
      }
    }

}
```

3) Esquemas algorítmicos

3.a) (2 puntos) Una fábrica tiene un número de pedidos pendientes por fabricar. Cada pedido se caracteriza por el número de días que dura su fabricación y por la contaminación total que produce.

La ley limita la contaminación que se puede producir en cada intervalo de 10 días consecutivos.

Dado un conjunto de pedidos, se pide implementar un algoritmo basado en "Vuelta Atrás" que permita determinar si es posible o no realizar dicho conjunto de pedidos sin superar los límites de contaminación impuestos por la ley. En caso de que sea posible realizar todos los pedidos sin superar los límites de contaminación, el algoritmo deberá retornar un orden de fabricación de los pedidos que permita alcanzar el objetivo.

3.b) (1 punto) Diseñar e implementar un algoritmo voraz que permita conocer el mayor número de trabajos que es posible realizar sin superar los límites de contaminación y el orden en el que habría que realizarlos. Detallar también los cambios realizados en la clase `Pedido` (en caso de que sea necesario realizar alguno). Explicar con lenguaje natural el criterio de selección utilizado.

3.c) (0.5 puntos) Indicar si el algoritmo voraz implementado permite obtener la solución óptima en todos los casos o, si por el contrario, se trata de un heurístico. Demostrar (de

manera informal) que se trata de un algoritmo óptimo o presentar un contraejemplo que demuestre que no siempre encuentra la solución óptima.

Para implementar los algoritmos solicitados, se dispone de la clases `Pedido` y `SolParcial` ya implementadas.

Código de la clase `Pedido`:

```
/**
 * Pedido realizado a la fábrica
 */
public class Pedido {

  /**
   * Días necesarios para producir el pedido
   */
  private int díasProducción;

  /**
   * Contaminación producida en la fabricación del pedido
   */
  private double contaminación;

  /**
   * Crea un pedido
   * @param díasProducción días necesarios para producir el pedido
   * @param contaminación contaminación producida en la fabricación
   * del pedido
   */
  public Pedido(int díasProducción, double contaminación) {
    this.díasProducción = díasProducción;
    this.contaminación = contaminación;
  }

  /**
   * Retorna los días necesarios para producir el pedido
   * @return días necesarios para producir el pedido
   */
  public int díasProducción() {
    return díasProducción;
  }

  /**
   * Retorna la contaminación producida en la fabricación del
   * pedido
   * @return contaminación producida en la fabricación del pedido
   */
  public double contaminaciónTotal() {
    return contaminación;
  }
```

Interfaz de la clase `SolParcial`:

`public SolParcial(Pedido[] pedidos)`

Construye una solución parcial vacía (sin ningún pedido incluido)

Parameters:
pedidos - pedidos que tiene la empresa

`public boolean factible(Pedido pedido)`

Indica si la solución parcial sigue siendo factible (no supera los límites de contaminación en el intervalo de control) si se le añade el pedido indicado

Parameters:
pedido - pedido a añadir a la solución actual
Returns:
true si se puede añadir el pedido sin superar los límites de contaminación y false en caso contrario

`public void añadePedido(Pedido pedido)`

Añade el pedido a la solución parcial

Parameters:
pedido - pedido a añadir

`public void eliminaPedido(Pedido pedido)`

Elimina el pedido de la solución actual

Parameters:
pedido - pedido a eliminar

`public boolean pedidoIncluido(Pedido pedido)`

Indica si el pedido está ya incluido en la solución parcial

Parameters:
pedido - pedido a comprobar si está incluido
Returns:
true si el pedido ya está incluido en la solución parcial y false en caso contrario

`public int numPedidosIncluidos()`

Retorna el número de pedidos incluidos en la solución parcial

```
Returns:
    número de pedidos incluidos en la solución parcial
```

`public Pedido[] ordenDeFabricación()`

```
Retorna un array con los pedidos incluidos en la solución parcial
en el orden en el que fueron añadidos

Returns:
    array con los pedidos incluidos en la solución parcial en el
    orden en el que fueron añadidos
```

Solución 3.a:

```
public class ResuelveVA {

  /**
   * Resuelve el problema de la posibilidad de realizar un conjunto
   * de pedidos sin superar los límites de contaminación. Utiliza
   * un algoritmo basado en "Vuelta Atrás"
   * @param pedidos conjunto de pedidos a fabricar
   * @return null si no es posible fabricar los pedidos sin superar
   * los límites de contaminación o un orden de realización de los
   * pedidos que permite realizarles sin superar los límites
   */
  public static Pedido[] resuelve(Pedido[] pedidos) {

    // solución inicial vacía (sin ningún pedido incluido)
    SolParcial sol = new SolParcial(pedidos);

    // primera llamada al algoritmo recursivo
    boolean encontradaSolución = resuelveRec(pedidos, sol);

    if (encontradaSolución)
      return sol.ordenDeFabricación();
    else
      return null;
  }


  /**
   * Algoritmo recursivo basado en vuelta atrás que resuelve el
   * problema de la contaminación de un conjunto de pedidos
   * @param pedidos conjunto de pedidos a fabricar
   * @param sol solución parcial compuesta por un conjunto de
   * pedidos que no superan los límites de contaminación
   * @return true si ha encontrado la solución. En ese caso sol
   * será la solución al problema (incluye todos los pedidos sin
   * superar los límites de contaminación). Retorna false si no se
   * ha encontrado solución.
   */
```

```
  private static boolean resuelveRec(Pedido[] pedidos,
                                     SolParcial sol) {
    // finaliza si la solución ya contiene todos los pedidos
    if (sol.numPedidosIncluidos() == pedidos.length) {
      // solución encontrada: finaliza
      return true;
    }

    // prueba añadiendo cada uno de los pedidos
    for(Pedido p: pedidos) {

      // si el pedido ya está incluido en la solución parcial o su
      // inclusión hace que la solución no sea factible, pasa al
      // siguiente pedido
      if (sol.pedidoIncluido(p) || !sol.factible(p))
        continue; // posibilidad de ramificación no válida

      // añade el pedido a la solución y realiza la llamada
      // recursiva
      sol.añadePedido(p);
      boolean encontradaSolución = resuelveRec(pedidos, sol);

      // finaliza si ha encontrado la solución
      if (encontradaSolución)
        return true;

      // si con el último pedido añadido no ha encontrado la
      // solución, le saca y prueba con el siguiente
      sol.eliminaPedido(p);
    }

    return false; // solución no encontrada
  }
}
```

Solución 3.b:

El criterio de selección utilizado consiste en ir eligiendo los pedidos de menor a mayor contaminación por día. Otros criterios de selección sencillos serían, por ejemplo, elegir los pedidos de menor a mayor duración o de menor a mayor contaminación. Ninguno de ellos permite encontrar la solución óptima en todos los casos.

```
import java.util.Arrays;

public class ResuelveHeurístico {

  /**
   * Heurístico que busca el mayor número de pedidos que es
   * posible realizar sin superar los límites de contaminación.
   * El criterio de selección utilizado consiste en ir eligiendo
   * los pedidos de menor a mayor contaminación por día.
   * No siempre encuentra la solución óptima.
```

```
     * @param pedidos conjunto de pedidos a fabricar
     * @return orden de realización de pedios encontrado
     */
    public static Pedido[] resuelve(Pedido[] pedidos) {
      SolParcial sol = new SolParcial(pedidos);

      // ordena el array de pedidos de menor a mayor contaminación
      // por día
      Arrays.sort(pedidos);

      // va metiendo pedios a la solución hasta que no se
      // puedan meter más
      for(Pedido p: pedidos) {
        if (!sol.factible(p)) {
          // no se pueden seguir metiendo pedidos
          return sol.ordenDeFabricación();
        }

        // añade el pedido y pasa al siguiente
        sol.añadePedido(p);
      }

      // retorna el mejor orden de fabricación encontrado
      return sol.ordenDeFabricación();
    }
}
```

En la clase `Pedido` hay que añadir "`implements Comparable<Pedido>`" y el siguiente método `compareTo`:

```
  /**
   * Compara los pedidos en base a su contaminación por día
   * @param pedido objeto a comparar con el actual
   */
  @Override
  public int compareTo(Pedido p) {
    return (int)(contaminación/díasProducción
                 - p.contaminación/p.díasProducción);
  }
```

Solución 3.c:

El criterio de selección elegido (seleccionar los pedidos de menor a mayor contaminación por día) no permite encontrar la solución óptima en todos los caos. Por lo tanto, el algoritmo desarrollado es un heurístico que en la mayoría de los casos permitirá encontrar buenas soluciones, pero que no asegura encontrar la óptima.

Contraejemplo:

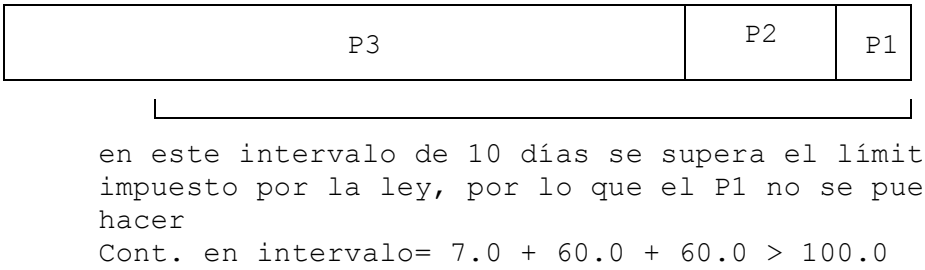
Supongamos que el límite de contaminación que la ley permite en cada intervalo de 10 días consecutivos es 100.0 y que la empresa tiene 3 pedidos:

P1: (días:1, cont: 60.0, cont/día: 60.0)

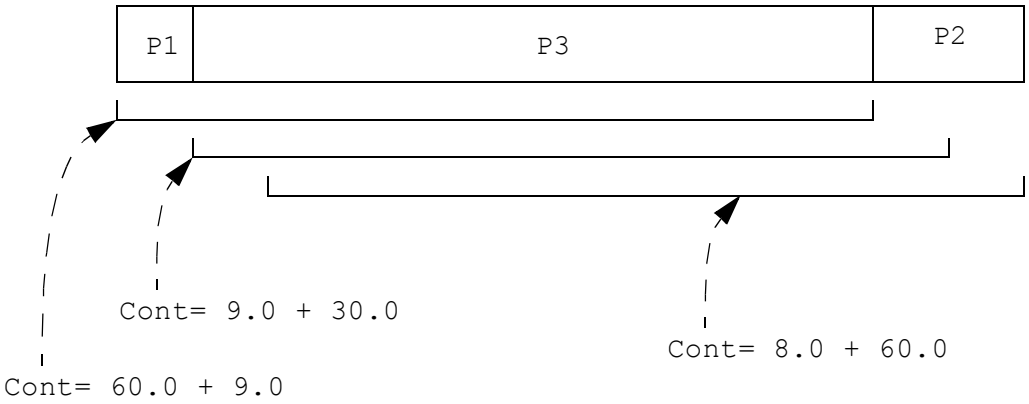
P2: (días:2, cont: 60.0, cont/día: 30.0)

P3: (días:9, cont: 9.0, cont/día: 1.0)

Con nuestro criterio de selección haríamos primero el pedido 3, luego el 2 y después trataríamos de hacer el 1, pero no sería posible ya que se superaría la contaminación máxima:



Sin embargo la solución óptima (NO encontrada por nuestro algoritmo) sería:



En la que no hay ningún intervalo de 10 días en el que se supere el límite impuesto por la ley, por lo que con esta ordenación de los pedidos se podrían realizar los 3, y no sólo 2 como obtuvimos con nuestro heurístico.

4) (0.6 puntos) Escribir una función Haskell (junto con su cabecera) que reciba un elemento de cualquier tipo y una lista de elementos de ese mismo tipo y retorne 1 si el elemento está en la lista y 0 en caso contrario. Ejemplos de invocación

```
elemEnLista 1 [2,4,3,-2,5] ----------------------------> 0
elemEnLista (3,4) [(8,-2),(3,4),(3,6)] ----------------> 1
```

(0.4 puntos) Utilizar la función anterior para escribir otra función (y su cabecera) que reciba dos listas del mismo tipo de elementos y genere la lista que contiene los elementos comunes a ambas listas. Ejemplo de invocación:

```
elemComunes [3.5,2.8,-0.5] [-0.5,1.1,2.8,4.3] ----->[2.8,-0.5]
```

Solución:

```
-- El elemento está en la lista
-- Recibe un elemento de cualquier tipo y una lista de elementos de
-- ese mismo tipo y retorna 1 si el elemento está en la lista y 0 en
-- caso contrario
elemEnLista  :: (Eq a) => a -> [a] -> Int
elemEnLista _ []   = 0
elemEnLista e (x:xs)
   | e == x        = 1
   | otherwise     = elemEnLista e xs

-- Intersección de listas
-- Recibe dos listas del mismo tipo de elementos y genera la lista
-- que contiene los elementos comunes a ambas listas
elemComunes     ::  (Eq a) => [a] -> [a] -> [a]
elemComunes  _  []    =  []
elemComunes  [] _     =  []
elemComunes (x:xs) l
   | elemEnLista x l == 0    =  elemComunes xs l
   | otherwise               =  x:elemComunes xs l
```